An Evaluation Platform for Forensic Memory Acquisition Software

By

Stefan Voemel and Johannes Stuttgen

Contents lists available at SciVerse ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

An evaluation platform for forensic memory acquisition software

Stefan Vömel*, Johannes Stüttgen

Department of Computer Science, Friedrich-Alexander University of Erlangen-Nuremberg, Martensstraße 3, 91058 Erlangen, Germany

Keywords:
Memory forensics
Memory acquisition
Live forensics
Evaluation
Metrics
Correctness
Atomicity
Integrity of a memory snapshot
Forensic soundness

A B S T R A C T

Memory forensics has gradually moved into the focus of researchers and practitioners alike in recent years. With an increasing effort to extract valuable information from a snapshot of a computer's RAM, the necessity to properly assess the respective solutions rises as well. In this paper, we present an evaluation platform for forensic memory acquisition software. The platform is capable of measuring distinct factors that determine the quality of a generated memory image, specifically its *correctness*, *atomicity*, and *integrity*. Tests are performed for three popular open source applications, *win32dd*, *WinPMEM*, and *mdd*, as well as for different memory sizes.

1. Introduction

Memory forensics, i.e., the forensic acquisition and analysis of volatile data in system RAM, has evolved from a niche discipline to an important field in computer forensics over the last years. This development is driven by the need to find suitable alternatives to classic, persistent data-oriented approaches that are more and more incapable of sufficiently addressing various recent phenomena. For instance, while storage capabilities of hard drives are rapidly growing, respective I/O bandwidths are not (Patterson, 2004; Roussev and Richard III, 2004). As a consequence, producing forensic duplicates of the respective media, and completing an investigation in time gets more and more cumbersome. Additionally, the use of file or full disk encryption technologies can quickly make evidence extraction and examination infeasible in case a suspect is unwilling to share the corresponding decryption key. Last but not least, various malicious applications are known to execute in memory only and do not leave any persistent traces on the hard disk of a user any longer (Moore et al., 2003; Sparks and Butler, 2005). Failing to consider all relevant sources of an incident may thus create an incomplete picture of a situation and lead to wrong conclusions.

As Vömel and Freiling (2012) have pointed out, a lot of work in the area of memory forensics has been done concerning *analysis*-related aspects to date. The acquisition and image generation process, on the other hand, has been researched to a lesser degree. In particular, the quality and performance of existing solutions has not been adequately assessed yet, even though various authors have argued that especially software-based acquisition methods solely produced so-called "fuzzy" (*smeared*) snapshots due to concurrent system activity (Libster and Kornblum, 2008).

Related Work. Literature on the quality or impact of memory acquisition solutions is still sparse. A general, "ideal" method for duplicating volatile information with respect to forensic investigations is sketched by Schatz (2007, p. S128). Such a method produces an image of physical RAM that "is a precise copy [of] the original host's memory" and is "available, working on arbitrary computers (or devices), and additionally must be reliable, either producing a trustworthy result or none at all". Even though Schatz outlines these requirements in more detail in the

* Corresponding author.
E-mail addresses: stefan.voemel@cs.fau.de (S. Vömel), johannes.stuettgen@cs.fau.de (J. Stüttgen).

remainder of his paper, his explanations rather aim at illustrating the limitations of current technologies than at finding accurate metrics for sound memory imaging.

Walters and Petroni (2007) attempt to estimate the degree of system contamination that is caused by acquisition-related activities. For this purpose, they create snapshot "baselines" and match the system state before and after an utility has been launched. Similarly, Sutherland et al. (2008) pursue an empirical approach and observe the system state over the time a memory snapshot is generated. The main objective of their study is to assess common imaging programs by monitoring modifications of certain system resources, e.g., the Windows registry or the hard disk partition. Lempereur et al. (2010, p. 3) argue however, that "[w]hile their results and method can serve as useful guidelines for the developers and users of forensic software, the selection of metrics [is] arbitrary and may not represent the best way to quantify the impact of forensic tools". As an alternative, they propose comparing "the state of a machine on which forensic acquisition has been performed, to the state of an identical, unaltered, machine". To illustrate the practicability of their approach, the authors set up several virtual machines and correlate the results of the individual test runs with the help of a Python-based framework.

Su and Wang (2011) present a statistical model for calculating the probability a memory area is changed when loading an imaging program into RAM. A similar methodology is pursued by Savoldi et al. (2010) in order to quantify the degree of "uncertainty" due to executing an imaging solution on a machine. Although such concepts are helpful in fostering a better understanding of the challenges in forensic memory acquisition, they only aid little in recommending a specific solution in practice.

In contrast, Inoue et al. (2011) describe a visualization technique that is capable of revealing systematic errors in memory images. Based on so-called graphical *dotplots*, products can be assessed upon their *correctness*, *completeness*, *speed* as well as their *amount of interference*. In a later work, Vömel and Freiling (2012) have shown how these metrics can be reduced to the three fundamental criteria *correctness*, *atomicity*, and *integrity*. They also illustrate how other previously established terms in memory acquisition theory coined by Schatz (2007) can be mapped and integrated into their model. However, while the individual factors are precisely defined and formalized, it is only vaguely pointed out how they can be actually measured.

Contributions of the Paper and Results. In this paper, we present the architecture of an evaluation platform that helps quantify the different metrics outlined by Vömel and Freiling (2012). Thus, for the first time, we are able to determine the degree of *correctness*, *atomicity*, and *integrity* of memory acquisition utilities in an in-depth and repeatable manner.

We hope that by *measuring* instead of estimating factors for sound memory imaging, investigators have a starting point for comparing available products more reasonably and better appraise their individual strengths and weaknesses. For instance, the following questions can be answered with the help of our approach:

- Does an acquisition utility produce a snapshot that equals the size of the physical address space?
- Does the created snapshot contain the data that was stored in a memory page at the time said page was imaged?
- How does an acquisition utility cope with errors and areas of memory that cannot be accessed?
- In how far does concurrent activity interfere with the imaging process?
- What is the impact of an acquisition utility, and how much memory is changed when the application is loaded into RAM?
- What is the amount of memory that is changed in the course of the imaging period?

To assess the quality of an acquisition solution, the software is executed within a guest operating system on top of a highly customized version of the *Bochs* x86 PC emulator (The Bochs Project, 2013a) that serves as the foundation for our evaluation platform. We pursue a *white-box testing* methodology, i.e., we inspect the source code of the utility and slightly adapt it to our needs. Specifically, we insert a number of *hypercalls* that inform our platform of important system events and operations, e.g., the point of time when the acquisition process is initiated/finished or when a page of RAM is about to be duplicated. The platform intercepts the individual hypercalls, creates a protocol of the different activities, and generates its own view of the system state. This view is matched with the produced memory snapshot in a later analysis phase to derive the value of the described metrics.

Our approach is applicable to all memory acquisition applications for which access to source code is given. We exemplify this by focusing on Microsoft Windows systems (Linux or other operating systems are not supported by our platform at the time of this writing) and present first results for three popular acquisition utilities, *win32dd* (Suiche, 2009b), *WinPMEM* (Cohen, 2012), and *mdd* (ManTech CSI, Inc., 2009). We show that two products initially produced forensic snapshots that differed both in size and contents. In more detail, the respective solutions ignored regions of memory that were used by hardware devices. As a consequence, the offset mapping of subsequent memory areas was corrupted, leading to potential data misinterpretations in the course of the investigation phase.

A second observation we made in our evaluation was that, in dependence of the time that is required for the imaging process, maintaining the atomicity and integrity of a snapshot gets increasingly more difficult, even on idle systems. With our platform, the degree of such consistency violations can be estimated. Last but not least, our experiments again highlight the need for rigourously assessing the functionality and performance of forensic applications. Only then can investigators reasonably justify and explain the use of a particular method.

Outline of the Paper. The remainder of this paper is outlined as follows: In Section 2, we briefly review the three fundamental criteria for sound memory imaging formalized by Vömel and Freiling (2012). In Section 3, we describe our evaluation methodology as well as the architecture of the developed testing platform. A study of three software-based acquisition utilities and their corresponding performance results are subject of Section 4. In Section 5, we explain the advantages and disadvantages of our evaluation approach and discuss possible alternatives. In addition, we outline a number of weaknesses and limitations our platform has to cope with that still need to be addressed in the future. We conclude with a short summary of our work and indicate several opportunities for further research in Section 6.

2. Criteria for sound memory acquisition

In the following, we give a brief overview of important criteria required for sound memory acquisition as they have been described by Vömel and Freiling (2012).

Correctness. The notion of correctness refers to the values stored in the produced image file. Precisely, a forensic memory snapshot is considered *correct with respect to a set of memory regions* $R \subseteq \mathscr{R}$[1] if "for all these regions, the value that is captured in the snapshot matches the value that is stored in this region at this specific point of time" (Vömel and Freiling, 2012, p. 131). Intuitively, a memory acquisition approach thus works *correctly with respect to these regions* if it images the exact values that were saved in memory at the time the snapshot was taken. The *degree of correctness* is the percentage of memory regions that have been acquired correctly.

Even though correctness is seemingly a trivial requirement at first glance, Vömel and Freiling stress its importance for the process of forensically-sound memory imaging: On a compromised system, for instance, malicious software may attempt to manipulate or impede the individual acquisition operations. Likewise, an imaging solution may contain software errors, so that the layout of the physical address space is inadequately mapped to the image file, thereby causing a potential later misinterpretation of the respective data. We will see examples for such errors in Section 4.2.

Taking these aspects into consideration, failing to ensure the correctness of a generated snapshot may therefore create a false "ground truth" for a subsequent investigation and potentially put the entire chain of custody at stake.

Atomicity. While a forensically-sound memory snapshot should only contain "true" values according to the previous requirement, the criterion of atomicity stipulates that the memory image should neither be affected by signs of concurrent activity. Vömel and Freiling (2012) depict such activities with the help of two-dimensional *space-time diagrams* that visualize the dependency and interrelationship of operations on memory regions across time. Thereby, a memory snapshot is modeled as a so-called *cut* through the space–time diagram of a system, in accordance with the respective terms used in distributed systems theory (Lamport, 1978; Mattern, 1989). If a snapshot is atomic, the corresponding cut must be *consistent*, i.e., program causality is correctly represented in the image file. The *degree of atomicity* is the percentage of memory regions that satisfy consistency in this respect.

Of course, software-based memory acquisition programs commonly produce snapshots with less than 100% atomicity, because the system state can effectively not be frozen during the imaging process. As a result, the generated snapshot becomes "fuzzy" (Libster and Kornblum, 2008, p. 14), and we will illustrate how such causality violations can be measured for several utilities and memory sizes in Section 4.2.

Integrity. With the help of the third and last criterion, it is possible to observe the state of memory over a certain time period. Generally, a snapshot satisfies *integrity with respect to a specific point of time* τ if "the values of the [acquired] memory regions that are retrieved and written out by an acquisition algorithm have not been modified after τ" (Vömel and Freiling, 2012, p. 132). The *degree of integrity* is the percentage of memory regions that are unaltered after τ. As the authors argue, τ preferably represents a point of time very early in the investigation process when the state of memory has not yet been affected by acquisition-related activities. If this state of memory remains unchanged until it is saved to the image file, the integrity of the respective regions is maintained. It is important to emphasize that in this case, the memory regions are also correctly and atomically duplicated.[2]

Measuring the level of integrity permits assessing the impact of a given acquisition approach on a computer's RAM. For instance, by loading a software-based imaging utility into memory, specific regions are irrevocably overwritten, and the degree of system contamination increases. As we will see in Section 4.2, while the impact of the evaluated solutions is generally negligible, considerable portions of memory are modified in the course of the imaging process, even on idle systems.

In the following, we will present our measurement methodology and the architecture of our testing platform.

3. Measurement methodology and platform architecture

As mentioned in the introduction of this paper, our evaluation is based on a *white-box testing* approach, i.e., we examine the source code of the desired imaging software, identify the relevant instructions that are responsible for reading and writing out portions of memory to the respective image file, and slightly adapt the code to our needs by inserting a number of *hypercalls*. The different hypercalls are intercepted and processed by our platform in the course of the imaging period to create an external view of the system state. By matching this view with the produced memory snapshot, its level of *correctness*, *atomicity*, and *integrity* can later be determined.

Before we describe the design, functionality, and mode of operation of our platform in more detail in Sections 3.2

[1] $\mathscr{R}$ refers to the set of all addressable memory regions.

[2] Please note, however, that satisfying correctness and atomicity does not necessarily imply the integrity of the respective memory regions (see Vömel and Freiling, 2012).

and 3.3, we briefly illustrate the memory acquisition process as it is frequently implemented in forensic applications for Microsoft Windows operating systems.

3.1. Forensic memory imaging on Microsoft Windows systems

A common technique for acquiring a (raw) forensic copy of a computer's RAM relies on leveraging the `\\.\Device\PhysicalMemory` section object that, as the name suggests, provides access to sections of physical memory. For security reasons, however, permissions to open the resource in user space were revoked with the introduction of Microsoft Windows Server 2003 (Service Pack 1, see Microsoft Corporation, 2013a). For this reason, all of the imaging applications we considered for our evaluation did not only consist of a user-mode administration program, but also of a kernel-level driver. The latter typically calls the `ZwOpenSection` function in the first step to retrieve a handle to the `\\.\Device\PhysicalMemory` object. After determining the actual size of memory, portions of RAM may then be read out page wise, for instance, with the help of the `ZwMapViewOfSection` function. In the last step, a mapped section can either be directly written to the image file in kernel space or transferred to user space via a buffer for further processing.

An illustration of a typical imaging algorithm is depicted in Fig. 1. As can be seen, the design of an acquisition program can be quite simple in practice. However, errors may particularly be introduced if the size of the physical address space is not correctly calculated, or the return values of called functions are not rigourously tested. We will discuss an example for the former case in a later section of this paper. Milković (2012), on the other hand, demonstrates how imaging operations can be easily blocked or evaded with a number of simple anti-forensic techniques. With regard to this, it is important to emphasize that the procedure of invoking the `\\.\Device\\PhysicalMemory` object may also be susceptible to manipulation (see Bilby, 2006). Therefore, more sophisticated solutions frequently support alternative acquisition methods as well, e.g., by calling kernel routines such as `MmMapIOSpace` (Microsoft Corporation, 2013b). For the sake of simplicity, these approaches have not yet been included in our evaluation though.

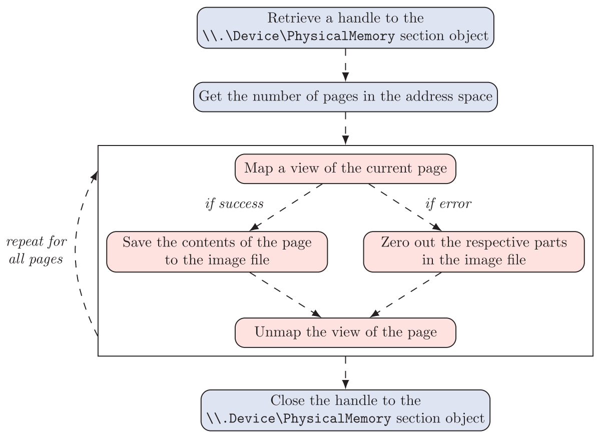
Retrieve a handle to the \\.\Device\PhysicalMemory section object
Get the number of pages in the address space
Map a view of the current page
if success
if error
Save the contents of the page to the image file
Zero out the respective parts in the image file
Unmap the view of the page
repeat for all pages
Close the handle to the \\.\Device\PhysicalMemory section object

Fig. 1. Sample algorithm for acquiring a forensic copy of memory.

3.2. Platform architecture

As we have already indicated, our testing platform is built upon *Bochs*, an open source x86 PC emulator (The Bochs Project, 2013a). Bochs' distinct advantages over similar products such as *QEMU* are its smaller code base by roughly 50% (~250,000 lines of code) and the possibility to implement the stub of a custom *instrumentation interface* comparatively easily. Our interface is largely written in C, apart from performance-critical parts that were developed in assembly language, and provides a series of callback functions that are invoked when specific system events and operations occur. In particular, with the help of the `bx_instr_lin_access` function, we are able to monitor linear memory accesses of the guest system. We will see in a later section of this paper that this capability is beneficial for determining the degree of atomicity.

As we have also pointed out already, the guest system can communicate with the emulator via a number of *hypercalls* that are triggered in the case of imaging-related activities. Technically, a hypercall uses the `EAX` register to indicate a specific event and pass additional meta information to the instrumentation interface if required. By issuing a *breakpoint* command (`INT 3`), the respective hypercall is executed in the next step. We have patched Bochs such that the interrupt is intercepted and gets solely processed by our interface. Before control is returned to the guest operating system, the vector is discarded. Thereby, the operation appears as completely transparent to the guest. Due to this behavior, applications running on the emulated machine can no longer be debugged. We believe that this loss of functionality is acceptable in the scope of our evaluation though.

An overview and short description of the hypercalls we have defined is given in Table 1. As can be seen, several calls are reserved for administrative purposes, e.g., for externally shutting down and resetting the guest system or for signaling the size of the physical address space to Bochs (see also Section 4.2). On the other hand, most hypercalls are directly related to the imaging process. For instance, the `HCALL_LOAD_ACQUISITION_SOFTWARE` instruction is invoked by a small wrapper program in order to mark the point of time before the acquisition program is loaded into memory. We will illustrate in the following section how, based on the individual hypercalls, the different factors for sound memory imaging can be measured.

3.3. Measuring factors for sound memory imaging

Correctness. In order to determine the correctness of an acquisition solution, we create a view of the guest's physical address space as it is seen by Bochs, in parallel to the imaging process. This external view is then matched with the produced snapshot in a later analysis phase to identify possible deviations. Specifically, when an imager accesses a memory page for subsequent duplication (e.g., via the `ZwMapViewOfSection` function), the respective page number is signaled to the instrumentation interface. Our platform then creates an external copy of the memory region the page is stored in before passing control back to the guest system. Please note that while all external operations

Table 1
List of hypercalls processed by the instrumentation interface.

Hypercall	Description
HCALL_BEGIN_LOG	Signals the beginning of the memory acquisition process.
HCALL_END_LOG	Signals the end of the memory acquisition process.
HCALL_BEGIN_IMG_PAGE	Signals the beginning of an image operation to the platform for each page.
HCALL_END_IMG_PAGE	Signals the end of an image operation to the platform for each page.
HCALL_LOAD_ACQUISITION_SOFTWARE	Notifies the platform of the point of time shortly before the memory acquisition program is loaded into memory.
HCALL_END_ACQUISITION_SOFTWARE	Notifies the platform of the point of time the acquisition program has completed its operations and is unloaded from memory.
HCALL_QUIT_SIM	Notifies the platform that the imaging process has finished and that the guest should be shut down.
HCALL_LOAD_IMAGER_CONFIG	Notifies the platform that more information about a certain acquisition program is requested for a test run.
HCALL_SIGNAL_PAGE_SIZE	Notifies the platform of the page size.
HCALL_SIGNAL_NUMBER_OF_PAGES	Notifies the platform of the number of memory pages.

are atomically executed, mapping a page inside the guest system and informing the host platform of this process are generally not, because we wanted to reduce modifications of the original imaging code to a minimum. Consequently, there is a small time frame in which a page may be updated, and the respective changes are reflected in the memory snapshot of the host system but not of the acquisition utility (or vice versa). In each of our experiments, these differences comprised only a few bytes though and are therefore negligible in our opinion.

In sum, with the help of the described method, we are able to verify whether the size of a forensic memory snapshot equals the size of the physical address space as well as whether the data stored in the image file corresponds to the contents of memory at the time the snapshot was taken.

Atomicity. Measuring the level of atomicity a forensic memory image satisfies was the most complex and challenging task in our evaluation. For performance reasons, we pursued an *indirect* approach and attempted to quantify the degree of atomicity *violations*. Such a violation occurs when a memory region that has already been imaged is accessed by a running thread, and, based on this access, another memory region is modified that still needs to be duplicated. In this case, the snapshot becomes inconsistent, because an effect is reflected in the image file without including the corresponding cause. In the space–time diagram illustrated in Fig. 2, for instance, a concurrent thread (as indicated by the black circles) accesses the memory regions r_1, r_2, r_3, r_4 while the acquisition program (as indicated by the black squares) is being executed. As can be seen, the thread accesses the memory region r_1 at time t_1 *after* the region has been imaged, and, in the following, manipulates memory regions r_2 and r_3 at times t_2 and t_3, although these are not yet represented in the snapshot and will be written out at a later time. Consequently, with respect to the regions r_2 and r_3, the atomicity criterion is violated.[3]

In practice, it is hard to verify whether the memory operations of a thread are truly *causally related*, i.e., whether changing the value of one memory region (directly) leads to the change of another region. With respect to *all* threads running on a machine, this task is nearly infeasible, otherwise we would have to taint track the entire system. For this reason, we measure a slightly weaker metric, namely the degree of *potential atomicity violation*, i.e., atomicity violations that are unspecified whether they would effectively lead to a modification of the respective program flow. As such, the metric is an *upper bound* for the degree of atomicity violation. The complement of this value, on the other hand, represents the absolute and accurate *minimum* degree of atomicity a snapshot satisfies.

To quantify the amount of potential atomicity violations, we hook the `bx_instr_lin_access` callback function (see Section 3.2) to observe all memory operations once the imaging procedure has started and keep track of the pages that have already been acquired. If a thread accesses a page after it has been imaged, it is inserted into a self-balancing binary search tree that serves as a *watchlist* for potential atomicity violators. The currently running thread is identified with one of two methods, depending on the processor privilege level (*ring 0* or *ring 3*) the corresponding process is executed on: For user-level threads, we obtain a pointer to the *Thread Environment Block* (TEB) that is referenced at an offset of the `FS` segment register (Schreiber, 2001). The TEB, in turn, points to a `_CLIENT_ID` structure that stores the unique thread identification number. In contrast, for kernel-level threads, we obtain the corresponding pointer from a substructure of the *Kernel*

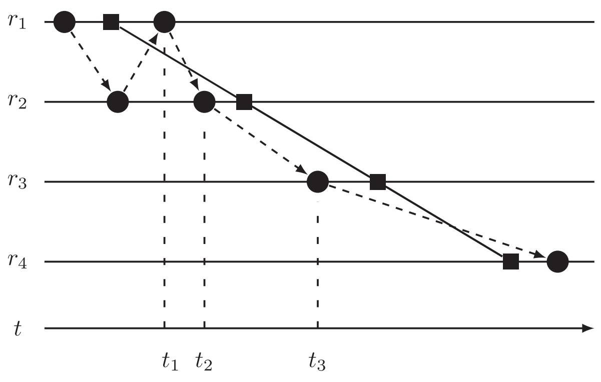

Fig. 2. Example of an atomicity violation.

[3] Technically, the atomicity criterion is violated with respect to memory region r_4 as well. This violation is of no relevance in the example, however, because the memory region has already been acquired at the time the violation occurs.

Processor Control Region (KPCR). The KPCR stores processor-related information and is accessible over the FS segment register as well (Schreiber, 2001). It is important to keep in mind that all referenced addresses are virtual though. Thus, to process the respective data within Bochs, a virtual-to-physical address translation process has to be completed first for each operation.

When a thread performs a write operation on a memory region that has not been imaged yet, it is checked whether the corresponding thread identification number is included in the watchlist of potentially atomicity-violating candidates. If this is the case, the operation is logged, using a custom binary logger we have implemented for this task. With the help of the generated log file, the upper bound for the degree of inconsistency in the memory image can then be estimated in the next step.

Integrity. Similarly to the process of measuring the correctness of a memory snapshot, we determine its level of integrity by creating copies of the physical address space at several points in time that are later matched in an analysis phase. In more detail, we duplicate the state of memory of the guest operating system shortly before the acquisition program is loaded into RAM as well as after the imaging process has finished. The former state is defined to represent the state of memory at time τ, in correspondence to our explanations outlined in Section 2.

To signal the respective point of times, we trigger the `HCALL_LOAD_ACQUISTION_SOFTWARE` and `HCALL_END_LOG` hypercalls that are intercepted by our instrumentation interface (see Section 3.2). For the first task, a small wrapper utility is running on the guest machine.

When the hypercalls are received, the platform temporarily suspends the execution of the guest system and creates a raw memory snapshot. An additional third snapshot is taken before the actual imaging process is started, i.e., *after* the respective solution has been loaded into RAM, but *before* the first page has been acquired. Thus, by comparing the snapshot taken at time τ with the state of memory at the end of the acquisition process, we can quantify the amount of memory that has either changed over time or remained stable. Likewise, by matching the images taken at time τ and at the beginning of the acquisition process, we can roughly estimate the impact of the imaging solution on the system, i.e., contents of RAM that have been overwritten by launching the respective program files. Please note, however, that this metric is only meaningful when other influencing factors such as the level of concurrent activity have been minimized. It is therefore planned in the future to find better approaches for measuring the impact of an acquisition utility.

We have conducted several experiments to quantify the correctness, atomicity, and integrity of forensic memory snapshots created by three different imaging applications. A summary of these activities is subject of the following section.

4. Evaluation

4.1. Evaluation methodology

We have evaluated the performance and quality of three popular memory acquisition utilities, namely *win32dd* (Suiche, 2009b), *WinPMEM* (Cohen, 2012), and *mdd* (ManTech CSI, Inc., 2009). While all applications have originally been open source, win32dd has been replaced by a closed source variant and is only distributed as part of the *MoonSols Windows Memory Toolkit* to date (Suiche, 2013). We were therefore only able to test the last publicly available version of the source code (`v1.2.1.20090106`).

All experiments were run on a standard off-the-shelf computer system with an Intel i5-650 processor and 8 GB of RAM. The only addition was a solid state drive that was used by the binary logger for performance reasons. For our platform, we reserved a maximum of 2 GB of physical memory. For the emulated machine, we chose memory sizes between 512 MB and 2 GB. The latter value represents the maximum amount of memory that is supported by Bochs at the time of this writing, a fact that had unfortunately not been documented in the corresponding user manual and was only discovered by the authors after development of the platform had been finished. As the main guest system, we set up a default installation of Microsoft Windows XP (SP3). To simulate a standard computer as best as possible, the list and configuration of system services was not adapted. However, all tests initially started from an *idle* state, i.e., after the startup process of the operating system and all applications had been fully completed.

For each memory size we considered in our evaluation (`512 MB, 1024 MB, 2048 MB`), we prepared 30 system *snapshot templates*. Such a snapshot template was created by temporarily suspending the guest system, a feature that is available over the Bochs graphical interface and that permits saving the current state of the processor, memory, and attached devices to hard disk (The Bochs Project, 2013b). Due to programming errors in the emulating engine, however, user input and output are not processed any longer when the simulation is resumed at a later time. For this reason, all image-related activities were automated without requiring any further manual intervention.

For each snapshot template, a special hypercall from the host to the guest system indicated the imaging solution to assess. Once a test run had been completed, the snapshot was reverted, and the simulation was restarted from the initial system state for the remaining evaluation candidates. As such, the performance results of the different products with respect to a specific snapshot template were directly comparable. In total, we conducted 270 experiments (90 per tested RAM size) for the three acquisition utilities. In dependence of the available memory capacity, between 4.37 GB and 12.37 GB of free hard disk space were required for the created log file.[4] Taking the externally and internally generated snapshots into account as well (5 in total), about 6.87 GB–22.37 GB of free space were therefore needed for every experiment.

[4] An entry in the log file occupied 12 bytes, i.e., roughly between 391 million (for 512 MB of RAM) and 1107 billion (for 2048 MB of RAM) atomicity-related memory operations were observed on average during an imaging process.

4.2. Results

Correctness. The physical address space of a computer is not only used by the operating system and executed applications, but also shared by hardware devices for *memory-mapped I/O* (MMIO) operations. When generating a forensic snapshot, these areas of memory need to be taken into consideration, too. Specifically, an acquisition solution should identify and zero out MMIO regions in the image file if the respective addresses cannot be accessed. The recommended method for the first task is invoking the undocumented `MmGetPhysicalMemoryRanges` function that indicates the memory structure as it is seen by Windows (Russinovich, 1999). In our tests, both win32dd and mdd initially determined the size of the address space incorrectly. For instance, mdd calculates the size of memory (in pages) based on the output of the `GlobalMemoryStatusEx` function which, unfortunately, does not include information about the MMIO space. Consequently, when iterating through the address space, these regions are ignored, and a smaller snapshot is produced. What is worse, because the image file is written sequentially instead of logically, the offset mapping of subsequently accessible memory areas is corrupted. For analysis techniques that rely on offset interpretation in the evidence extraction phase of the investigation, this may be a significant problem. With respect to *win32dd*, the error has been fixed in later (closed source) versions of the software (see Suiche, 2009a), the publicly available version of *mdd* is, however, still affected. In order to adequately test both products, we have therefore developed patches that address the issue, so that all considered solutions eventually produced snapshots that truly equaled the size of memory. One peculiarity of Bochs is, however, that the top of the guest's physical address space is always shortened by 16 pages (65,536 bytes). Thus, for a machine with 512 MB of memory for instance, the total number of pages corresponds to 131,056 (536,805,376 bytes) instead to 131,072 (536,870,912 bytes). We believe this may be due to some internal configuration, contacting a Bochs developer did not help shed light on this issue yet though.

The results of our experiments are presented in Table A1 of the appendix.[5] As can be seen, all imaging utilities were proven to achieve a correctness rate over 99.3%, i.e., the solutions are capable of reliably acquiring the contents of memory. For a small number of pages (see Rows 2, 6, and 10 of Table A1), the comparison of the generated snapshot with our external image indicated minor differences. As we have pointed out in Section 3.3, these differences lead back to little inaccuracies in our measurement approach, rather than to a malfunction in the imaging software. In most cases, the respective changes were limited to a few bytes per page (typically less than 20). For win32dd, however, more than 25% of one memory page was updated in each test run. We have not investigated the exact reason for this behavior yet, but we assume the location may be used for a program buffer.

[5] All results are average values, based on 30 test runs for each imager and per memory size.

If a memory region could not be read successfully, all utilities zeroed out the corresponding parts in the snapshot. Surprisingly, errors were not only signaled for the memory space of hardware devices as expected, but also for various pages of the operating system. Again, the exact reasons for this behavior still need to be examined in more detail. Likewise, it is still unclear at the time of this writing why the execution of WinPMEM led to a significantly higher number of read access violations (98–99 pages) in comparison to its competitors.

Atomicity. In our experiments, the level of potential atomicity violations rapidly increased with the size of installed memory (see Table A2 of the appendix). While we monitored potential snapshot inconsistencies between 37.29% and 39.55% of the pages on a machine with 512 MB of RAM, the number of violations jumped up to more than 75% for memory capacities of 2 GB. We believe that the major influencing factor for this steep rise is *time*, assuming the respective load on the system remains constant: As it takes imagers longer to complete their operations on computers with larger amounts of memory, concurrently running applications also have a longer time span to access areas that have already been duplicated and, in the next step, modify regions that are possibly not yet part of the snapshot. In short, with longer imaging periods, it gets obviously more and more difficult to keep the image file free from "smearing" (see also Fig. A1).

At the time of this writing, it is still an open research problem in what cases and in how far concurrent activity during the acquisition process actually has an impact on the "outcome" of a later analysis (see also Section 6). On the other hand, a significant degree of unatomicity is counterintuitive to classic perceptions of "forensic soundness" (see Ball (2005) and the discussion by Casey (2007)), and it is imaginable that the admissibility of evidence that is clearly based on a highly inconsistent source may be questioned in a trial. Investigators should keep these aspects in mind when using software utilities for the acquisition of a computer's RAM.

Integrity. In contrast to the degree of atomicity, the level of integrity the created snapshots satisfied *increased* with growing amounts of available memory. On average, between 27.373% and 28.587% of the data were subject to change in the course of the imaging period on a machine with 512 MB of RAM. Roughly about 47%–49% of all pages were affected. As opposed to this, on a system with 2 GB of memory, only between 17.905% and 18.447% of the data were modified, even though the number of affected pages slightly went up in most cases. This development was expected, because on a system with constant load but higher memory capacities, proportionally less amounts of space are required for the operating system and running applications. In comparison to former tests by other authors for alternative products and approaches (Walters and Petroni, 2007; Schatz, 2007), our results are similar. For higher capacities, the level of change is smaller, but with approximately one fifth of the size of the address space still significant. This effect is likely to aggravate again, however, once concurrent activity gets more intense, and the system load reaches its peak.

Regarding the impact of an acquisition utility, between 0.87 and 1.33 MB of memory are changed after loading the respective solution into RAM. As we have pointed out in

Section 3.3, these values represent an *upper bound* though, rather than an accurate estimation. In sum, however, all tested applications seem quite light-weight and reduce external dependencies on other software components to a minimum.

A summary of our experiments and the corresponding results are listed in Table A3 of the appendix. The percentage of bytes that remained unchanged in the course of the imaging period is depicted for the different products and memory sizes in Fig. A2.

5. Discussion

5.1. Black-box vs. white–box testing

As we have already explained, our evaluation platform relies on a *white-box testing* approach, i.e., we have to slightly adapt the source code of the acquisition program and insert several hypercalls that indicate specific events such as the beginning and end of the imaging process. Unfortunately, various solutions are closed source and are frequently commercially distributed. Prominent examples include *Memoryze* (Mandiant, 2011), *FastDump Pro* (HBGary, 2013), or *FTK Imager* (AccessData, 2012). With respect to such applications, we have initially attempted to pursue a *black-box testing* approach, i.e., measuring the quality of an utility despite lacking explicit knowledge of its internal mode of operation. For this purpose, we used inline code overwriting techniques to transfer execution to small code caves identified in the respective binary files. The code caves contained instructions to trigger a hypercall and initiate communication with the instrumentation interface. However, this method is unreliable for several reasons: First, it is unsure whether sufficiently large areas of free space can be found for the hypercall instructions in every case. Second, in order to get accurate measurement results, it is necessary to insert two hypercalls preferably close to the acquisition routine. This, in turn, would require reverse engineering of the executable though, a process that can be quite cumbersome and is usually legally prohibited.

We also attempted to reduce the number of hypercalls and recognize the imaging process solely based on memory access patterns. In practice, however, this procedure did not prove useful either. We therefore believe that our current implementation of white box testing is best suited for our needs, even though we were capable of evaluating only a small number of products available on the market to date so far. We hope, however, that further vendors will provide us with the source code of their solutions in the future, so that these utilities can be assessed as well.

5.2. Limitations of the platform

Our platform still has to deal with a number of weaknesses and limitations at the time of this writing: First, due to some internal program structures, the platform is unfortunately only able to evaluate 32-bit applications. In addition, using Bochs as the underlying emulation engine restricts the size of memory to 2 GB. This is a significant limitation, as even RAM sizes in modern desktop computers are frequently larger, and it would be particularly interesting to assess imagers with respect to the 4 GB boundary.

As we have also pointed out, while we can measure the correctness of a memory image quite accurately, we are only able to quantify the minimum level of atomicity a snapshot satisfies based on the upper bound of potential atomicity violations. Likewise, we can only roughly estimate the impact of the acquisition program by matching the state of memory at time τ with the state of memory shortly before beginning an imaging operation. For the latter metric, a more suited alternative thus needs to be found.

5.3. Operational capabilities of memory acquisition software

In our experiments, all tested memory acquisition utilities showed a similar performance. As a consequence, forensic investigators may choose the product they find most appealing for their work, assuming the default technique of generating a raw snapshot via the `\\.\Device\PhysicalMemory` section object is not subverted by malicious software, and more sophisticated imaging methods, e.g., via the `MmMapIOSpace` function (see Section 3.1), are not required. On the other hand, while acquisition utilities can likely be used in most scenarios (including incident response), a potentially high degree of atomicity and integrity violations has to be accepted.

Thus, for computer systems that run within a controlled environment and that can be comfortably configured before an incident is likely to happen, using the built-in *CrashOnCtrlScroll* functionality of the operating system may be a viable alternative (Microsoft Corporation, 2011). The function permits saving the state of memory as well as of relevant processor registers to hard disk when a specific keyboard shortcut is pressed. Because system interrupts are unmasked before generating the snapshot, the impact on the level of atomicity is lower. However, the feature is disabled by default though and needs to be manually activated. Furthermore, as drivers have the possibility of receiving a notification about an impending imaging operation via callback routines such as `KeRegisterBugCheckCallback` (see Russinovich et al., 2009, p. 1120), malicious application have a short timeframe for unloading themselves before the crash dump is taken. Investigators should therefore carefully judge the benefits and drawbacks of this method.

Wang et al. (2011) illustrate an approach for reliably generating a snapshot of memory by leveraging the processor's System Management Mode (SMM). In this mode, the execution of the operating system is temporarily paused, and the system state can be examined. In contrast, Martignoni et al. (2010) as well as Yu et al. (2012) present an imaging concept that is based on hardware virtualization. To the best of our knowledge, the latter projects are more of academic nature though and have not been extensively tested in practice yet.

6. Conclusion

We have presented a platform for evaluating forensic memory acquisition software with respect to the three factors *correctness*, *atomicity*, and *integrity*. These factors determine the quality of a RAM snapshot and are measured using a *white-box testing* approach, i.e., we must be provided with the source code of the respective solution. With the help of a number of *hypercalls* that are inserted close to image-related code parts, we can then signal important events and operations. A highly customized version of the x86 emulator *Bochs* intercepts these notifications and creates a protocol of the imaging process.

In a preliminary study, we have assessed the performance of three acquisition applications, namely *win32dd*, *WinPMEM*, and *mdd*, for memory sizes between 512 MB and 2 GB. For this task, we have analyzed 270 snapshots of systems in an idle state. Our study revealed that not all products were initially capable of generating a copy of the entire physical address space. Even worse, the affected solutions produced image files with mismatching data offsets. Because an analysis of these files would possibly lead to false results in a later investigation, we have patched the different utilities so that the problem was fixed, and correct snapshots were eventually generated.

One interesting observation we made in our evaluation was that the level of atomicity decreased with growing memory sizes. We argued that with larger amounts of memory and, thus, a longer time that is needed to complete the acquisition process, keeping the image file free of inconsistencies gets more and more difficult due to concurrent activity. In contrast, on a system with constant load, proportionally less areas of memory are subject to change, and the level of integrity a snapshots satisfies increases.

In sum, the performance of the tested acquisition utilities (after fixing the respective programming errors) did not significantly differ. This is not surprising, because all solutions internally access the `\\.\Device\PhysicalMemory` section object in kernel space to create a copy of the volatile storage. As we have pointed out in Section 3.1 though, these operations may be intercepted by malicious software to either prevent imaging completely or present a modified view of system RAM. For our experiments, we have assumed the operating system of the machine is not compromised. It is also important to note, however, that both win32dd as well as WinPMEM offer alternative imaging methods that can circumvent anti-forensic attempts. It is planned to evaluate the quality of these methods in the future, too.

Opportunities for Future Research. So far, all of our tests were initiated from an idle system state. For assessing the quality of an acquisition utility more extensively, it is necessary to simulate different system loads and, in particular, monitor the level of atomicity and integrity changes. Experiments should also be conducted on various operating systems, including both server and desktop versions, and on systems that have deliberately been infected with malware.

Using Bochs as the basis for our solution was a suboptimal choice in retrospect. Even though we were able to implement a rudimentary instrumentation interface quite quickly, customizing the emulator to our needs and creating a working prototype of the platform took significant efforts. This is mainly due to poor user documentation that is incomplete in most parts and only very marginally covers technically more complex concepts. In addition, the readability of the source code is greatly affected by a vast number of references to program macros. As it is even stated in the official developer manual, many macros have "inscrutable names", and "[o]ne might even go as far as to say that Bochs is macro infested", so that "too much stuff happens behind the programmer's back" (The Bochs Project, 2013c). Last but not least, the functionality of suspending and resuming a simulation is deeply flawed internally. For this reason, many of our original approaches for running a memory acquisition test had to be redesigned. For better long-term maintainability, we therefore recommend to port the platform to a more stable and mature environment such as *QEMU* (Bellard, 2012).

As we have seen, the level of atomicity and integrity a snapshot satisfies can be significantly influenced due to concurrent activity in the course of the imaging period. While violations of these factors may be in direct contradiction to classic perceptions of "forensic soundness" (see Section 4.2), the actual consequences of such violations with respect to a subsequent investigation are still mostly unclear. For example, it has yet to be found out in what cases and at what level inconsistency leads to significantly different analysis results. Finding answers to these questions will be an interesting field of research in the near future.

Acknowledgments

We would like to thank Felix Freiling and Thomas Schreck for reading a previous version of this paper and giving us valuable feedback and suggestions for improvement.

Appendix A. Results of the evaluation

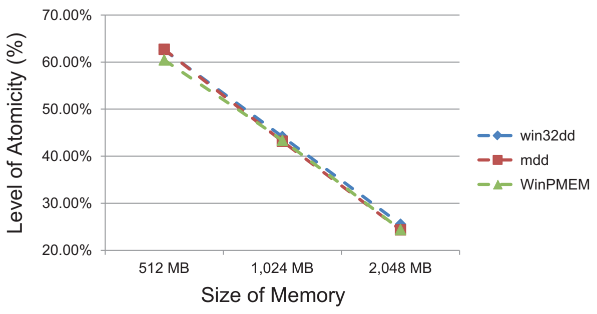

Fig. A1. Level of atomicity achieved by different memory acquisition applications.

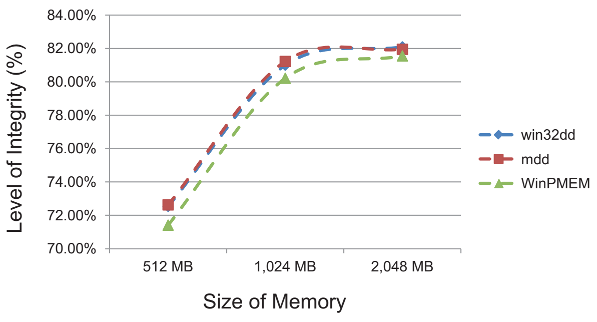

Fig. A2. Level of integrity achieved by different memory acquisition applications.

Table A1
Results for the correctness evaluation of different memory acquisition applications.

		512 MB	1024 MB	2048 MB
Number of pages in the address space		131,056	262,128	524,272
win32dd	Number of correctly imaged pages	130,269 (99.40%)	260,572 (99.41%)	521,183 (99.41%)
	Number of pages with observed differences	19	20	17
	Number of bytes changed in total	1446	1484	1388
	Number of indicated read errors (incl. MMIO)	768	1536	3072
mdd	Number of correctly imaged pages	130,286 (99.41%)	260,590 (99.41%)	521,198 (99.41%)
	Number of pages with observed differences	1	1	1
	Number of bytes changed in total	1074	1075	1076
	Number of indicated read errors (incl. MMIO)	769	1537	3073
WinPMEM	Number of correctly imaged pages	130,188 (99.34%)	260,492 (99.38%)	521,100 (99.39%)
	Number of pages with observed differences	1	1	1
	Number of bytes changed in total	14	14	14
	Number of indicated read errors (incl. MMIO)	867	1635	3171

Table A2
Results for the atomicity evaluation of different memory acquisition applications.

		512 MB	1024 MB	2048 MB
Number of pages in the address space		131,056	262,128	524,272
win32dd	Number of pages unaffected by concurrent activity	81,979 (62.55%)	115,763 (44.16%)	133,871 (25.54%)
	Number of pages affected by potential atomicity violations	49,077 (37.45%)	146,365 (55.84%)	390,401 (74.46%)
mdd	Number of pages unaffected by concurrent activity	82,180 (62.71%)	113,199 (43.18%)	127,648 (24.35%)
	Number of pages affected by potential atomicity violations	48,876 (37.29%)	148,929 (56.82%)	396,624 (75.65%)
WinPMEM	Number of pages unaffected by concurrent activity	79,217 (60.45%)	113,697 (43.37%)	127,964 (24.41%)
	Number of pages affected by potential atomicity violations	51,839 (39.55%)	148,431 (56.63%)	396,308 (75.59%)

Table A3
Results for the integrity evaluation of different memory acquisition applications.

		512 MB	1024 MB	2048 MB
Number of pages in the address space		131,056	262,128	524,272
win32dd	Number of pages remaining consistent over the imaging process	68,842 (52.53%)	138,407 (52.80%)	265,586 (50.66%)
	Number of pages changed during the imaging process	62,214 (47.47%)	123,721 (47.20%)	258,686 (49.34%)
	Bytes of memory remaining consistent over the imaging process	389,382,288 (72.537%)	869,675,703 (80.999%)	1,762,918,568 (82.095%)
	Bytes of memory changed during the imaging process	147,423,088 (27.463%)	204,000,585 (19.001%)	384,499,544 (17.905%)
	Number of pages changed after loading the acquisition program	1361	1325	1362
	Bytes of memory changed after loading the acquisition program	990,792	924,691	1,024,498
mdd	Number of pages remaining consistent over the imaging process	69,625 (53.13%)	140,621 (53.65%)	266,684 (50.87%)
	Number of pages changed during the imaging process	61,431 (46.87%)	121,507 (46.35%)	257,588 (49.13%)
	Bytes of memory remaining consistent over the imaging process	389,867,381 (72.627%)	872,074,311 (81.223%)	1,759,805,581 (81.950%)
	Bytes of memory changed during the imaging process	146,937,995 (27.373%)	201,601,977 (18.777%)	387,612,531 (18.050%)
	Number of pages changed after loading the acquisition program	1964	1977	1969
	Bytes of memory changed after loading the acquisition program	1,287,561	1,288,320	1,399,336
WinPMEM	Number of pages remaining consistent over the imaging process	66,305 (50.59%)	137,669 (52.52%)	266,588 (50.85%)
	Number of pages changed during the imaging process	64,751 (49.41%)	124,459 (47.48%)	257,684 (49.15%)
	Bytes of memory remaining consistent over the imaging process	383,347,294 (71.413%)	861,257,118 (80.216%)	1,751,276,845 (81.553%)
	Bytes of memory changed during the imaging process	153,458,082 (28.587%)	212,419,170 (19.784%)	396,141,267 (18.447%)
	Number of pages changed after loading the acquisition program	1338	1307	1341
	Bytes of memory changed after loading the acquisition program	992,730	911,888	992,381

References

AccessData. FTK imager. https://ad-pdf.s3.amazonaws.com/Imager_3.1.2_RN.pdf; 2012.

Ball C. 6 on forensics – six articles on computer forensics for lawyers. http://www.craigball.com/articles.html; 2005.

Bellard F. QEMU. http://www.qemu.org/; 2012.

Bilby D. Low down and dirty: anti-forensic rootkits. In: Proceedings of Ruxcon 2006.

Casey E. What does “forensically sound” really mean? Digital Investigation 2007;4(2):49–50.

Cohen M. WinPMEM. http://scudette.blogspot.de/2012/11/the-pmem-memory-acquisition-suite.html; 2012.

HBGary. FastDump – a memory acquisition tool. http://www.hbgary.com/fastdump-pro; 2013.

Inoue H, Adelstein F, Joyce RA. Visualization in testing a volatile memory forensic tool. Digital Investigation 2011;8(1):S42–51.

Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 1978;21(7):558–65.

Lempereur B, Merabti M, Shi Q. Pypette: a framework for the automated evaluation of live digital forensic techniques. http://www.cms.livjm.ac.uk/pgnet2010/MakeCD/Papers/2010073.pdf; 2010.

Libster E, Kornblum JD. A proposal for an integrated memory acquisition mechanism. ACM SIGOPS Operating Systems Review 2008;42(3):14–20.

Mandiant. Memoryze. http://www.mandiant.com/products/free_software/memoryze/; 2011.

ManTech CSI, Inc. Memory DD. http://sourceforge.net/projects/mdd/files/; 2009.

Martignoni L, Fattori A, Paleari R, Cavallaro L. Live and trustworthy forensic analysis of commodity production systems. In: Proceedings of the 13th International conference on recent advances in intrusion detection (RAID) 2010.

Mattern F. Virtual time and global states of distributed systems. In: Workshop on parallel and distributed algorithms 1989.

Microsoft Corporation. Windows feature lets you generate a memory dump file by using the keyboard. http://support.microsoft.com/?scid=kb%3Ben-us%3B244139&x=5&y=9; 2011.

Microsoft Corporation. Device\PhysicalMemory object. http://technet.microsoft.com/en-us/library/cc787565%28v=ws.10%29.aspx; 2013a.

Microsoft Corporation. MmMapIoSpace routine (Windows drivers). http://msdn.microsoft.com/en-us/library/windows/hardware/ff554618%28v=vs.85%29.aspx; 2013b.

Milković L. Defeating Windows memory forensics. http://events.ccc.de/congress/2012/Fahrplan/events/5301.en.html; 2012.

Moore D, Paxson V, Savage S, Shannon C, Staniford S, Weaver N. Inside the Slammer Worm. IEEE Security and Privacy 2003;1(3):33–9.

Patterson DA. Latency lags bandwith. Communications of the ACM 2004;47(10):71–5.

Roussev V, Richard III GG. Breaking the performance wall: the case for distributed digital forensics. In: Proceedings of the digital forensic research Workshop (DFRWS) 2004.

Russinovich ME. MmGetPhysicalMemoryRanges function. The System Internals Newsletter 1999;1:5.

Russinovich ME, Solomon DA, Ionescu A. Microsoft Windows internals. 5th ed. Microsoft Press; 2009.

Savoldi A, Gubian P, Echizen I. Uncertainty in live forensics. In: Advances in digital forensics VI. IFIP Advances in information and communication technology. Boston: Springer; 2010. p. 171–84.

Schatz B. BodySnatcher: towards reliable volatile memory acquisition by software. Digital Investigation 2007;4:126–34.

Schreiber SB. Undocumented Windows 2000 secrets – a programmer’s cookbook. Addison Wesley; 2001.

Sparks S, Butler J. Shadow Walker – raising the bar for rootkit detection. http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf; 2005.

Su Z, Wang L. Evaluating the effect of loading forensic tools on the volatile memory for digital evidences. In: Seventh international conference on computational intelligence and security 2011.

Suiche M. Reply to HBGary. http://www.msuiche.net/2009/11/16/reply-to-hbgary-and-personal-notes/; 2009a.

Suiche M. Win32dd. http://www.msuiche.net/tools/win32dd-v1.2.1.20090106.zip; 2009b.

Suiche M. MoonSols Windows memory toolkit. http://moonsols.com/product; 2013.

Sutherland I, Evans J, Tryfonas T, Blyth A. Acquiring volatile operating system data tools and techniques. ACM SIGOPS Operating Systems Review 2008;42(3):65–73.

The Bochs Project. Bochs – the cross platform IA-32 emulator. http://bochs.sourceforge.net/; 2013a.

The Bochs Project. Save and restore simulation. http://bochs.sourceforge.net/doc/docbook/user/using-save-restore.html; 2013b.

The Bochs Project. Weird macros and other mysteries. http://bochs.sourceforge.net/doc/docbook/development/emulator-objects.html; 2013c.

Vömel S, Freiling FC. Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition. Digital Investigation 2012;9(2):125–37.

Walters A, Petroni NL. Volatools: integrating volatile memory forensics into the digital investigation process. In: Proceedings of Black Hat DC 2007.

Wang J, Zhang F, Sun K, Stavrou A. Firmware-assisted memory acquisition and analysis tools for digital forensics. In: Proceedings of the Sixth international workshop on systematic approaches to digital forensic engineering (SADFE) 2011.

Yu M, Lin Q, Li B, Qi Z, Guan H. Vis: virtualization enhanced live forensics acquisition for native system. Digital Investigation 2012;9(1):22–33.